Sections and Chapters

Gubert Farnsworth

Ngày 16 tháng 12 năm 2020

STRING MATCHING

1 String Matching Problem

Tìm kiếm chuỗi hiện diện trong cuộc sống, ví dụ như: tìm tên thầy dạy DSA trong danh sách giảng viên, tìm tên trong bảng điểm,.. hoặc trong khoa học, như là tìm liệu cấu trúc ADN của virus này có trong virus khác hay không,

2 String Matching Algorithms

1. Brute-force

   Giải thuật Brute-Force, hay còn gọi là vét cạn, là thuật toán đơn giản nhất trong các thuật toán tìm kiếm chuỗi con *pattern* trong chuỗi cha *text*.

   Có thể giải thích đơn giản, giải thuật Brute-Force so sánh lần lượt mỗi chuỗi con *subtext* của *text* có cùng chiều dài với *pattern* với *pattern*, nếu tìm được, trả về kết quả là vị trí được tìm thấy; khi không tìm được kết quả mong muốn, trả về giá trị quy ước là không tìm thấy.

   Trong ví dụ sau, ta sẽ làm rõ cách hoạt động của giải thuật này:
   text = `Let them go!`
   pattern = `them`

Let them go! them
Let them go! them
Let them go! them
Let␣them go! them
Let␣them go! them
Let them go! them
Let them go! them
Let them go! them
Let them go! them

Ta tìm thấy chuỗi pattern tại vị trí thứ 4!

Từ ví dụ trên, ta thiết kế mã giả cho giải thuật Brute-Force:

```
      vị trí tìm thấy = -1
      subtext = chuỗi con đầu text có độ dài bằng pattern
      while (chưa tìm thấy hoặc chưa tới cuối text)
        if (từng ký tự của subtext = pattern):
          trả về vị trí
        else:
          dịch chuyển chuỗi con subtext trong text sang phải 1 chữ cái
      Trả về: vị trí tìm thấy
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N
Phân tích độ phức tạp của thuật toán trong trường hợp xấu nhất:

- Mỗi lần so sánh với *subtext*, *pattern* phải so sánh nhiều nhất là M lần (trong trường hợp cả M - 1 ký tự đầu đều đúng).
- Có tất cả N - M + 1 chuỗi, vậy số chuỗi cần so sánh nhiều nhất là N - M + 1 *subtext* như vậy (trong trường hợp N - M + 2 chuỗi *subext* đầu không trùng với *pattern*)
  → Cần M(N - M + 1) lần. Vì duyệt tới cuối mảng nên đây là trường hợp tìm thấy ở cuối mảng, hoặc không tìm thấy
  → Cận trên $O(MN)$ (vì N > N - M + 1).
  → Cấp phát bộ nhớ: 0.

Phân tích độ phức tạp của thuật toán trong trường hợp tốt nhất:

- Trong trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*.
- Như vậy, chỉ cần tốn M lần so sánh các ký tự.
  → Cần M lần.
  → Cận trên $O(M)$.
  → Cấp phát bộ nhớ: 0.

Đánh giá:

- Dễ hiểu, thuật toán này chỉ duyệt từ đầu đến cuối, so sánh tuần tự từng chuỗi con với chuỗi cần tìm kiếm.
- Không cần bước tiền xử lý (như các thuật toán được trình bày bên dưới).
- Độ phức tạp $O(MN)$. Không cần xin thêm bộ nhớ.

2.1 Rabin-Karp

Thuật toán Rabin-Karp là một thuật toán được sử dụng để tìm kiếm chuỗi con *pattern* trong chuỗi cha *text* bằng cách sử dụng một hàm băm.

Hàm băm là một hàm chuyển đổi mọi chuỗi thành giá trị số, giá trị này được gọi là mã băm của nó. Ví dụ, chúng ta có thể có hàm băm hash("hello")=5.

Giống như Thuật toán Brute-Force, thuật toán Rabin-Karp cũng dịch *pattern* qua từng phần tử trong *text* để so sánh. Nhưng sự khác biệt là thuật toán Rabin-Karp so khớp mã băm của *pattern* với mã băm của chuỗi con *subtext* của *text*, và nếu mã băm khớp thì thuật toán sẽ so sánh từng ký tự trong 2 chuỗi với nhau.

Nếu mã băm được biểu diễn bằng số nguyên không quá 64 bits, độ phức tạp thời gian (time-complexity) của việc so sánh *pattern* có độ dài m với *subtext* có cùng độ dài giảm từ O(m) xuống O(1).

Tuy nhiên mọi thứ đều có hai mặt, vấn đề của hàm băm đó là mã băm của hai chuỗi khác nhau có thể bằng nhau. Ví dụ xét hàm băm hash(S) tính mã băm của xâu S bằng cách cộng mã ASCII của các kí tự trong S: hash("abcd")=97+98+99+100=394, hàm băm này quá đơn giản và có khả năng gây trùng mã băm cao: hash("dcba")=100+99+98+97=394, nhưng "abcd" $\neq$ "dcba".

Một hàm băm tốt thoả mãn các điều kiện sau:

- Tính toán nhanh.

- Xác suất trùng mã băm nhỏ.

Thuật toán Rabin-Karp xây dựng hàm băm với ý tưởng cơ số: xem mọi xâu như là một chuỗi số với một cơ số (base) nào đó. Hàm băm được tính tương tự như việc ta chuyển một số nguyên về giá trị của nó, nếu là xâu kí tự thì có thể sử dụng mã ASCII (hoặc UNICODE). Một số ví dụ:

- base=10, hash("425")=4×102+2×101+5×100=425.
- base=26, kí tự là chữ cái từ a đến z: hash("abc")=97×262+98×261+99×260=68219.

Để tránh tràn số thì kết quả trên được chia lấy dư cho một số q, thường được chọn là một số nguyên tố lớn. Nếu gọi tập các kí tự được sử dụng trong chuỗi là $\sum$ thì base thường được chọn sao cho base=$|\sum|$ hoặc là một số nguyên tố lớn.

Độ phức tạp thời gian để tính mã băm của chuỗi độ dài k mất O(k). Khi hiện thực thuật toán, ta sẽ "trượt" *pattern* có độ dài m trên *text* từ vị trí 0 đến n-m để so sánh mã băm. Rolling hash là hàm băm có thể tính mã băm h_i của *text*[i... i+m-1] dựa trên mã băm h_{i-1} của *text*[(i-1)... (i+m)] chỉ trong thời gian O(1) thay vì tính lại trong thời gian O(m), từ đó tăng tính hiệu quả.

$h_i = (base \times (h_{i-1} - base^{m-1} \times text[i-1]) + text[i+m-1])\%q$

Trong ví dụ sau, ta sẽ làm rõ cách hoạt động của giải thuật này:
Để đơn giản, ta chọn tập các kí tự được sử dụng trong chuỗi là {a, b, c, d, e, f, g, h, i, j} ứng với giá trị số lần lượt là {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} và base=10, q=13.
text = abccddaefg
pattern = cdd
Mã băm của *pattern*: hash("cdd")=$(3 \times 10^2 + 4 \times 10^1 + 4 \times 10^0)\%13 = 344\%13 = 6$

---

abccddaefg // hash("abc")=$(1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0)\%13 = 123\%13 = 6$
cdd // Vì mã băm của *subtext* "abc" và *pattern* bằng nhau
// nên thuật toán so sánh từng ký tự của chúng.
// Do "abc"≠*pattern* nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("bcc")=$(10 \times (123 - 10^2 \times 1) + 3)\%13 = 233\%13 = 12$
cdd // Vì mã băm của *subtext* "bcc" và *pattern* khác nhau
// nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("ccd")=$(10 \times (233 - 10^2 \times 2) + 4)\%13 = 334\%13 = 9$
cdd // Vì mã băm của *subtext* "ccd" và *pattern* khác nhau
// nên dịch *pattern* qua phải 1 phần tử để tiếp tục tìm kiếm.

---

abccddaefg // hash("cdd")=$(10 \times (334 - 10^2 \times 3) + 4)\%13 = 344\%13 = 6$
cdd // Vì mã băm của *subtext* "ccd" và *pattern* bằng nhau
// nên thuật toán so sánh từng ký tự của chúng.
// Do "abc"=*pattern* nên chuỗi đã được tìm thấy.

Từ ví dụ trên, ta thiết kế mã giả cho giải thuật Rabin-Karp:

```
function: Rabin-Karp String Matching
text: chuỗi cha
pattern: chuỗi cần tìm
subtext: 1 đoạn thân của chuỗi cha có cùng chiều dài với pattern.
Trả về: vị trí tìm thấy đầu tiên hoặc -1 nếu không tìm thấy.

  n=chiều dài chuỗi text
  m=chiều dài chuỗi pattern
  Tính mã băm của pattern và chuỗi con đầu text có độ dài bằng pattern
  Dịch pattern từ vị trí 0, qua từng phần tử của text đến vị trí n-m
    Kiểm tra mã băm của subtext hiện tại và pattern
      Nếu bằng nhau, so sánh từng ký tự trong 2 chuỗi
```

```
                Nếu hoàn toàn giống nhau, trả về vị trí
            Tính mã băm của subtext tiếp theo, nếu âm cộng thêm một lượng q
        Nếu trong vòng lặp không tìm thấy pattern, ta đã duyệt hết text, trả
            về -1 báo kết quả không tìm thấy pattern trong text.
```

Với chuỗi *pattern* có độ dài là M, chuỗi *text* có độ dài là N:
Phân tích độ phức tạp của thuật toán:

Trong quá trình tiền xử lý, để tính mã băm cho *pattern* cần duyệt qua M phần tử, tương tự với *subtext* đầu tiên *text*, vậy độ phức tạp là $O(M + M) \in O(M)$.

Trường hợp tốt nhất, có thể thấy *pattern* chính là *subtext* đầu tiên của *text*, khi đó chỉ cần tốn M lần so sánh các ký tự, kết hợp với quá trình tiền xử lý nên độ phức tạp là $O(M + M) \in O(M)$.

Trường hợp xấu nhất:

- Mỗi lần so sánh với *subtext*, *pattern* phải so sánh nhiều nhất là M lần (trong trường hợp mã băm bằng nhau và cả M - 1 ký tự đầu đều đúng).
- Có tất cả N - M + 1 chuỗi, vậy số chuỗi cần so sánh nhiều nhất là N - M + 1 *subtext* như vậy (trong trường hợp chuỗi cần tìm nằm cuối, N - M + 2 chuỗi *subext* đầu không trùng với *pattern*).
  → Cần nhiều nhất M(N - M + 1) lần so sánh. Vì duyệt tới cuối chuỗi nên đây là trường hợp tìm thấy ở cuối hoặc không tìm thấy và xảy ra trùng mã băm ở tất cả lần so sánh trước đó.
  → Cận trên $O(MN)$ (vì N > N - M + 1).

Do trong quá trình tìm kiếm có thực hiện tính toán và lưu mã băm nên chi phí bộ nhớ là hằng số $O(1)$.
Đánh giá:

- Mặc dù trong trường hợp xấu nhất độ phức tạp là $O(MN)$ không tốt hơn giải thuật Brute-Force, nhưng giải thuật Rabin-Karp hoạt động tốt hơn nhiều trong trường hợp trung bình và thực tế.
- Có bước tiền xử lý với độ phức tạp $O(M)$ trước khi bắt đầu tìm kiếm.
- Độ phức tạp $O(MN)$. Chi phí bộ nhớ $O(1)$.

PROGRAMING

(abc)

1 Introduce

2 Example Test